IMPLEMENTATION OF THE PAGE FAULT FREQUENCY REPLACEMENT ALGORITHM

Alexander Eugene Lancaster

NAVAL POSTGRADUATE SCHOOL

Monterey, California

THESIS

IMPLEMENTATION

OF THE

PAGE FAULT FREQUENCY

REPLACEMENT ALGORITHM

by

Alexander Eugene Lancaster, Jr.

Thesis Advisor: G. L. Barksdale, Jr.

June 73

Approved for public release; distribution unlimited.

Implementation of the Page Fault Frequency Replacement Algorithm

by

Alexander E. Lancaster, Jr.
Captain, United States Marine Corps
A.B., Columbia University, 1965

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
June 1973

ABSTRACT

This paper investigates the implementation of the page fault frequency (PFF) replacement algorithm as the mechanism for selecting and replacing pages of programs loaded into the main memory of a multiprocessing, multiprogrammed computer system. The frequency at which an executing program requires a page of virtual memory, the PFF, provides a basis for judging the real memory requirements of the program. Operating difficulties of PFF that reduce its usefulness in a time-shared computer system (Michigan Terminal System) are discussed, and a means of implementing the algorithm is proposed.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

The author wishes to acknowledge the invaluable assistance of his thesis advisor, Professor Gerald L. Barksdale, Jr., Chairman of the Computer Science Group, in the planning and the writing of this paper. Professor Barksdale's patience, understanding, and endurance were vital factors in its preparation. Furthermore, his ability to remain calm while the author plundered his personal library hastened the author's introduction into the wonders of systems programming.

Appreciation is extended to LTJG Raymond H. Bruebaker for his helpful suggestions in the preparation of the work. It was his course on operating systems that first interested the author in the subject.

Special thanks must also go to Kristina L. Bertwell, Parker G. Thomas, and Edwin V. Donnellan of the William R. Church Computer Center for their assistance and enthusiasm during the past two years of computing. Their sense of humor and professional competence helped the author through many trying hours.

I. INTRODUCTION

This paper investigates the implementation of the page fault frequency (PFF) replacement algorithm as the mechanism for selecting and replacing pages of programs loaded into the main memory of a multiprogrammed computer system. The problem is an exercise in resource allocation techniques that has significance in the overall functioning of a paged operating system. A review of the requirements for a demand-paged memory replacement scheme and an overview of a time-shared, multiprocessing, multiprogrammed computer system (Michigan Terminal System) provide a background for the analysis of PFF. Operating difficulties of PFF that reduce its usefulness are discussed, and a means of implementing the algorithm is proposed.

The concept of PFF was first proposed by Chu and Opderbeck[1] who examined its functioning in relation to two other popular replacement algorithms, least-recently used (LRU) and Denning's working set principle[2]. Using primarily mathematical analysis and stack processing [3], Chu and Opderbeck concluded that the frequency at which an executing program requires a page of code that is not in main memory, the page faulting frequency, could provide a basis for a replacement algorithm that is relatively independent of program behavior and input data. Though their work was limited to simulation experiments, they did provide an algorithm for using the PFF and a list of the required hardware mechanisms needed for implementation.

Their simulation showed that the PFF algorithm performed near the optimum when programs exhibited a high degree of execution locality. The experimental FORTRAN programs (both compile and execute) had this property, and a high percentage of the job stream at the Naval Postgraduate

School is FORTRAN. Furthermore, the small size of the main memory, 768K bytes, and the presence of only one drum store at the principle computer at NPS provides a realistic environment for the evaluation of the PFF algorithm functioning in a paged, time-sharing system.

II. MEMORY REPLACEMENT ALGORITHMS

The principal characteristic of time-sharing systems is conveyed in its very name; the system's resources are shared over time among many users in such a way that each user appears to have control and use of these resources exclusively. Since the cost in terms of money is high and the utilization in terms of time is generally low, it is considered inefficient to entirely give over the system to a single user for his work. Therefore, time-sharing systems attempt to multiplex resources among the users in some fashion in order to satisfy each within a reasonable period of time. These resources may be divided into two basic categories: software and hardware. By software resources, we mean such things as compilers for higher-level languages, assemblers, applications routines, and more basic things often called supervisor or monitor calls that enable the user to more easily use the machinery. Hardware resources generally are more visible; they are the card readers, printers, disk packs, etc., but principle among these are the processors and memory.

The heart of an operating system is the system supervisor, and almost every function performed by the supervisor involves the allocation of the system resources[4]. It is the task of the supervisor to provide the resources of the system to the user's process and thus enable some useful purpose to be served by the system's existence. Two seemingly contradictory goals appear in the functioning of the supervisor. First, since a time-sharing system usually has human beings "in the loop", it must minimize the amount of time spent waiting for a resource, and second, it must attempt to use all system resources efficiently[4]. In the formulation of a supervisor policy

for resource allocation, the basic decision is whether to preallocate all resources that a user may need during the execution of his program or allocate them only at the point at which they are required[5]. If the tasks of explicitly requesting an amount of main memory and indicating what is to be put into it are shifted from the process to the system, then the prerequisites for demand-paging are all present[5].

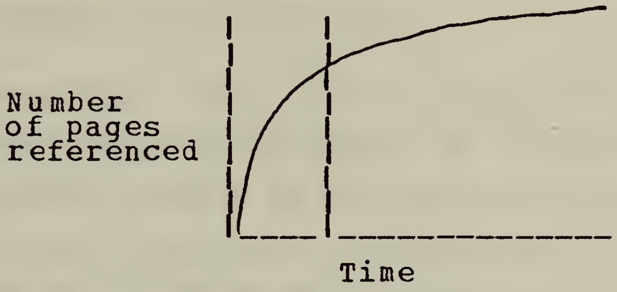
Demand-paging represents an approach to the sharing of main memory among portions of a process, called _pages_, and is generally taken to mean that these pages are brought into main memory _page frames_ (blocks) only upon the occurence of a page fault[7]; this mechanism is usually implicit and occurs whenever the process attempts to reference a page that is not in memory. The supervisor decides what page or pages are to be removed and when their removal is to be done in order to make room for the incoming page if the main memory is full at the time.

The fact that a page may not be in core at the time of its reference leads to the notion of _virtual processing time_: processing time spent in actual execution. Virtual processing time (VPT) may be considered to be real elapsed processing time (RPT), once execution is begun, less the amount of time spent waiting for pages to be read[1]. Clearly, VPT represents a lower bound to RPT, and it also represents a situation in which no time is spent waiting for pages to come into core. If the process can exceed main memory in size, it can be expected that some time will be spent by processes in these page waits. The supervisor naturally seeks to minimize this time since while in a page wait, the process is utilizing vital resources, mainly core memory. A replacement algorithm for a demand-paged system attempts to operate without _a priori_ knowledge of the program's page reference pattern. However, the page replacement algorithm may make assumptions about references the program is likely to make and try to avoid removing

those pages which are expected to be needed within a short period of time[6].

A. THE GENERAL PROBLEM

In a demand-paging environment, a program's paging behavior is not linear over time as shown in Figure 1.



Program Paging Behavior

Figure 1[7].

During the initial phase, the time between page faults is very short and page faults occur frequently. This might be termed the loading phase when the program is acquiring sufficient pages to make some meaningful execution possible. The second phase is characterized by a marked leveling of the page requests; the time between page faults becomes longer, so the program is now fully in meaningful execution[1,6,7]. The critical problem for any replacement algorithm is to allow the process to reach the point of meaningful execution quickly and sustain a useful time between page faults throughout execution.

Given that the VPT is not going to equal RPT and that pages are going to be moved into main memory only as required, it is reasonable to seek to bring a page into memory only once and leave it there until no longer required. The question is to determine which of a process's pages are no longer required at a given point in time while having on hand only information concerning its past behavior. Denning's optimal policy[6] is one that removes the page or pages that will be needed farthest in the

[illegible]

future. This serves to limit transfers and to leave in the memory those pages that the process will need shortly. Of the multitude of proposed algorithms, two are widely known, least-recently used (LRU) and Working Set. A third algorithm, page fault frequency (PFF), is the subject of this investigation.

B. THE LEAST RECENTLY USED ALGORITHM

The least-recently used (LRU) algorithm allows the processes in memory to acquire pages on demand until the number of available frames is near or at the limit of main memory. At this point pages are selected for removal from memory in order to increase the amount of available space; those selected for removal are the ones that have been referenced least since the last decision to deallocate pages was made. These pages are copied out to the auxiliary storage medium, the frames they formerly occupied are reclaimed by the system, and new pages are read into these frames when needed. It is a global algorithm; all pages are candidates for removal. Since there are economies of scale, typically more than one page is removed at a time since the cost in time and hardware resources to the system is substantial. Processes can interact and directly affect one another since a page fault on the part of one process may result in the ejection of a page belonging to another.

This can have significant implications; it may require that a process gaining control of a processor to initiate more page faults of its own in order to reacquire the pages necessary for continuing execution. In effect, the result has been to shorten the time between page faults and raise the paging rate of the system. This is particularly important in a system that employs a round-robin type of processor scheduling where each process is given a "slice" of processor time at regular intervals. As Chu and Opderbeck[1] have shown, the efficiency of this algorithm is

[illegible]

a function of the number of allocated page frames; the larger the amount of allocated space a process has, the more efficient the operation in terms of VPT and RPT. Determining the optimum amount of memory to give a process is difficult; giving too much memory results in inefficient utilization of the available page frames, while giving too little adds considerably to the paging traffic. The LRU algorithm is fairly simple to implement; the only addition to the hardware is a reference bit that is set whenever a reference is made to a block of memory, and most modern computing systems now have these.

C. THE WORKING SET PRINCIPLE

Denning's working set principle is predicated upon the notion of locality in programming. Within a given segment of time, the program in execution will reference a set of pages that usually does not change radically over time. This set of pages is the "working set." The size of the working set is a function of the time at which the determination is made and the length of the time segment. Ultimately, if the time segment length, **tau**, were equal to the RPT, then the entire process would comprise the working set, but as was intended, **tau** is very small, and the algorithm for replacement relies upon the fact that the relationship between the pages referenced within the last **tau** seconds and the entire program is non-linear. As execution of the process continues, the number of pages referenced within the last **tau** seconds varies. The replacement algorithm is executed not at the occurence of a page fault as with LRU, but every **tau** seconds. Those pages that have not been referenced will be removed from the main memory.

The parameter, **tau**, significantly affects the performance of the system utilizing the algorithm. The optimum value of this parameter can be expected to vary

between processes with detrimental effects to system operation. This particular algorithm also requires considerable modification to most existing hardware for implementation[8]. Programs with small working sets can be expected to perform well with the working set principle, but as the size increases, e.g. as in list-processing programs, performance deteriorates significantly with a fixed value of tau[1].

D. THE PAGE FAULT FREQUENCY ALGORITHM

The raticnale for the page faulting frequency algorithm begins with the graph of Figure 1. In the early phase of execution (the left part of the graph), the time between page faults is very short as the process gathers its pages for execution. Later in execution, the process' memory requirements will stabilize and slight changes in the amount of its memory will not affect the efficiency (VPT/RPT) significantly; this has been confirmed by Chu and Opderbeck[1] in simulation experiments. However, the system's performance is affected by these changes. If too much time in the system is devoted to paging operations, then fewer CPU cycles are available for execution of process instructions, yet unless a useful paging rate is maintained, the processes in the system cannot run to completion since all processing is stopped until the next pages are brought into memory.

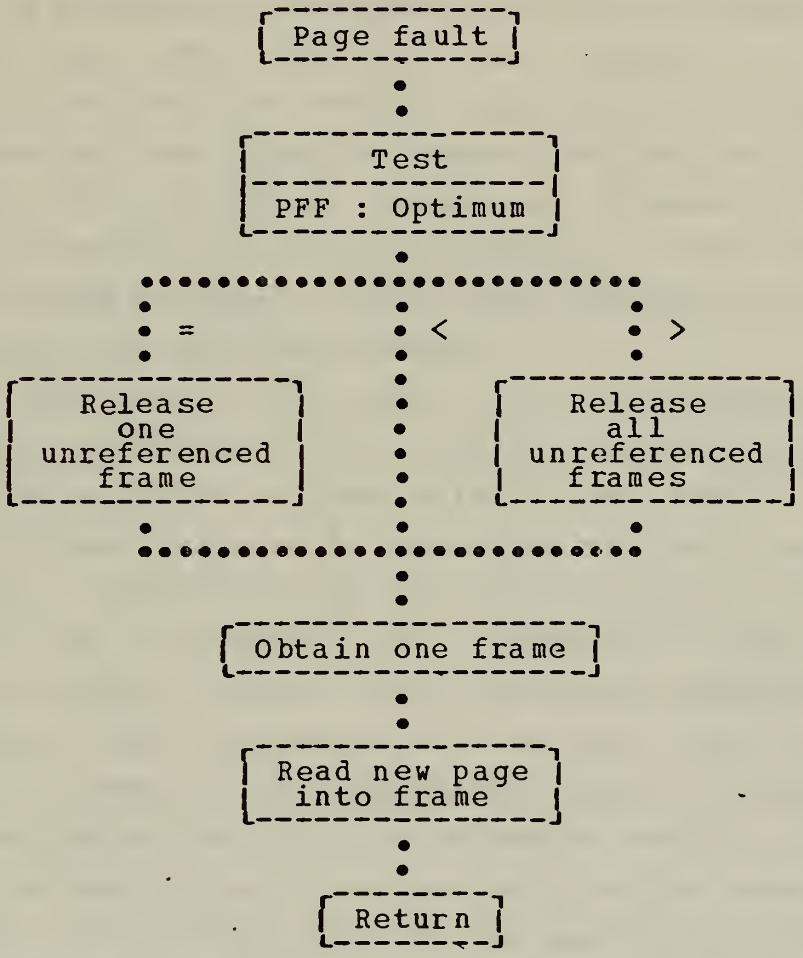
A system parameter called the page faulting frequency is used to balance the allocations of main memory. Chu and Opderbeck defined it as the total number of page faults caused by an executing process divided by its total number of page references, but in any demand-paging scheme, there is a minimum number of page faults that every process will cause, namely the number required to load each page or the total number of pages in the process. A further refinement, called the <u>normalized</u> PFF, can be made. Let TPF be the

[illegible]

total page faults caused by a process, NP be the number of pages in the process, and TNR be the number of changes in page references made by the process, then

$$\text{Normalized PFF} = (TPF - NP) / TNR.$$

This parameter is used to drive the decision process in the PFF memory replacement algorithm.



Page Fault Frequency Replacement Algorithm

Figure 2.

The algorithm, depicted in Figure 2, is based upon the system's maintenance of the optimum PFF for all executing processes. At the occurence of each page fault, the PFF is computed for the process; this is taken as a function of the time since the last page fault. If the time produces an optimum PFF, then the working set for the process is assumed to be in memory and no change to the amount of allocated

memory is made. If, however, the PFF is not optimal, then an adjustment is made to the number of allocated memory frames in an attempt to reestablish the optimum PFF. A high PFF, indicative of a low time between page faults, is taken to mean that the process is in its loading phase and needs more memory to effectively execute. In this case the process is allowed to accumulate pages and, hopefully, this will reduce its PFF back to the optimum. A low PFF indicates that the time between page faults is high, hence, the process has more than its needed working set in memory. Therefore, the allocated number of frames is reduced in order to drive the PFF up to the optimum. The amount that it is reduced is equal to the total number of unreferenced frames since that last page fault.

The algorithm is local in its allocation and deallocation decisions, but global in its mechanism to maintain the system's optimum value of the PFF. It combines some significant features of both the LRU and Working Set algorithms. First, it uses the LRU mechanism to reduce the allocation of a process, but applies it only to the executing process, hence, other processes cannot lose pages as a result. Here it appears to be a local LRU algorithm. Secondly, it operates on assumptions similar to those of the Working Set Principle, but the parameter **tau** is allowed to vary by executing the algorithm only on the occurence of a page fault. An optimum value of **tau** need not be known prior to execution of the process, but **tau** appears to self-determining as the process continues.

Performance of the PFF algorithm is more responsive to sudden changes in process memory requirements than LRU and less dependent upon program characteristics than the Working Set Principle. Chu and Opderbeck[1] reported in their simulation experiments that the algorithm is not particularly sensitive to the chosen value of the PFF, but gives good performance for those values close to the theoretical optimum. The algorithm is adaptive to the

changing conditions within an executing process, and it allows the system to remain responsive to process needs without degradation of overall performance.

Implementation of the PFF algorithm seems straightforward. Rather than attempt to use the PFF directly, the inverse, the time between page faults is used, and since this must be in VPT, a clock is needed to time the execution from one page fault to the next. This is normally available in most time-sharing machines and is a necessity for those that employ the time-slicing technique of processor scheduling. Some record is needed, though, of the time of execution at the last page fault to compare with the time of execution at the next page fault; the time between page faults is the difference between the two times. For removal of unreferenced pages, reference bits are used; hence, no special mechanism need be established nor any new hardware acquired.

III. SYSTEM DESCRIPTION

The vehicle chosen for this investigation of the PFF algorithm was the IBM System 360 Model 67 executing under the control of the University of Michigan Multiprogramming Supervisor/Michigan Terminal System (UMMPS/MTS), Version 2.2. This system employs the concept of virtual memory and features demand-paging as the principal mechanism for controlling memory allocation.

The Model 67 is the main computer in use at William R. Church Computer Center, Naval Postgraduate School. This installation is a duplex system and employs the full range of peripheral devices common to the general Model 67 system. The extent of the system main memory is normally 768K bytes (three modules), but it may be expanded to one megabyte. Auxiliary storage is primarily disk since only one IBM 2301 Drum Store is available[9].

A. THE HARDWARE SYSTEM

The Model 67 extends the capabilities of the previous Model 65 to suit the special needs of a time-sharing system which services a number of remote terminals[10,11]. In its basic form, the system is capable of addressing 2^{24} bytes of main storage, and in the extended form, 2^{32} bytes. The system can easily be configured for multiprocessing and allows for one or two central processing units to be included with each sharing a common main memory that can expand up to the limit of addressibility (eight core storage modules in the basic system). Each of the memory modules employs a lengthened version of the IBM physical storage protection key system to allow recording both the referencing of a physical storage block (2048 bytes) and the

occurence of a store operation within the block; hence, information is available within the storage key to indicate whether the block has been changed (due to a store operation) and whether the executing program has referenced the block since the last key setting. These features are critical to the operation of most memory replacement schemes. A high resolution timer is also included in each CPU to support the "time-slicing" technique of scheduling program execution[11].

Input/output operations are handled through an IBM 2846 Channel Controller which can be addressed by any of the functioning CPU's in the system. Control of I/O equipment is shared among the CPU's, thus enhancing the system's capability for parallel operation. A maximum of two channel controllers can be addressed by each CPU, and each channel controller can have up to seven channels attached. The channnel controller operates to establish a path between a CPU requesting a channel and the desired device. It appears, therefore, that any CPU has the capability to address any given device for I/O in the system. Further, the entirety of the system's main memory is available to a channel on a priority basis over any CPU; this allows operation of the I/O equipment independently of the central processors with little interference[12].

Segmentation is employed in this system as the means to realize virtual memory,but not with the full generality of segmentation as in MULTICS. Instead a variant, called linear segmentation by Watson[7], is provided. Here segment boundaries are not limits to sequential addresses; the first address in segment _n_ can be reached by indexing the last address of segment _n-1_, consequently, a segment translation step is not necessary in all cases of instruction execution. The Model 67 provides a maximum of 16 segments of processor storage available through 24-bit addressing. All segments are identical in size and consist of 256 pages of 4096 bytes each. The addition of the dynamic address translator (DAT)

with its eight associative registers allows efficient execution of programs that exceed the main memory capacity of the system. Each CPU has a DAT that functions to provide both dynamic relocation of program segments and a single-level addressing capability. This feature may be selectively disabled and the system run as the Model 65. Dynamic address translation is not available to the system's channels; all addresses required by the device must be main memory, i.e., absolute addresses[10,12].

B. THE SOFTWARE SYSTEM

The Michigan Terminal System, Version 2.2, is a multiprocessing, multiprogramming time-sharing system that currently supports both batch processing (in the traditional sense) and interactive or conversational terminal usage. MTS was designed originally for IBM/360 equipment and in later versions was modified to take full advantage of the special hardware and facilities of the Model 67[13]. The present version of MTS is capable of employing up to four processors as the computing resource of the system[14]. The organization of MTS makes it particularly amenable to modification since it is structured as a set of parallel processes with well-defined interfaces between them. Most of the routines communicate with few others, but nearly all do with the University of Michigan Multiprogramming Supervisor (UMMPS), hereafter called simply the supervisor[4].

The supervisor is the central program controlling the execution of all tasks (processes in MTS literature) the system. It may be accurately characterized as a set of subroutines that are activated by the hardware interrupt mechanism. The program that is the supervisor runs in the privileged state of execution, that is, all instructions on the machine can be executed, and the relocation hardware of the Model 67 is disabled during its execution; therefore,

[illegible]

all addresses are real and not subject to translation. Furthermore, all interrupts are disabled for the running of a supervisor subroutine, but the servicing of an interrupt is queued for execution upon leaving the subroutine unless it can be further delayed without degradation of the system's performance. In this system, the supervisor controls all physical input/output, processor and channel scheduling, and storage allocation.

Among the principal tasks that are run by the supervisor are MTS, HASP (Houston Automatic Spooling Program), and the Paging Drum Processor (PDP). MTS is a reentrant program which is activated once for each terminal that logs into the system and once for each batch task that enters. It provides the command language for the user and in general interfaces between the user and the supervisor/hardware mechanisms. Assemblers, compilers, subroutine libraries, graphics packages, etc. are called through MTS. HASP controls all spooling operations using card reader/punch equipment and printers; it initiates jobs submitted through any of the batch input streams. The PDP controls the reading and writing of any of the drum and disk storage systems in use by the supervisor for memory management. Pages that are removed by the supervisor to free memory frames for reallocation are sent to the PDP for storage on a drum and reread from there upon receipt of a page fault. Any overflow that may result from exceeding the capacity of a drum (maximum 900 pages per IBM 2301 drum) is spilled onto a disk back-up system.

The UMMPS/MTS system was chosen for the number of advantages it offered over other available systems that run on the Model 67 hardware, e.g. Time-Sharing System/360 (TSS) and Control Program-67 (CP-67). The independence of the components in the system makes it possible to replace or significantly modify one or more of them without affecting any other as long as the interfaces remain intact. The system control blocks and tables are accessible by few

programs, and in general, all system data and structures can be manipulated only by those programs designed specifically for that purpose. This is particularly important since the PFF algorithm requires storage of page fault times which would not normally be available in a system utilizing another algorithm. Reformatting of system control blocks is relatively simple. Two valuable features of the present system, the 'virtual Model 67' program and the built-in software monitor make the system especially useful for experimentation.

The virtual machine program allows the running of operating systems in their entirety as user programs without affecting system operation; this includes the ability to run MTS as a program in MTS and thereby debug any modifications prior to putting them into production and without the need for 'stand-alone' time on the entire system[13]. The software monitor program, called the data collection facility or DCF[15], makes it possible to enable and disable the selective collection of data regarding system operation for later analysis. Lastly, the software reliability of MTS has been reported as "good"[13].

IV. PROCESSOR AND MEMORY CONTROL IN UMMPS/MTS

The motivation for multiprogramming is to ensure that system resources are not allowed to remain idle due to the peculiarities of the processes in the system. If a running process blocks for some reason, e.g. an I/O operation, then another process is readily available in the system to begin using the CPU resource. The "time-slicing" technique of processor scheduling ensures that a process will block within a predictable period of time, namely the time-slice value, and free the CPU to begin executing another process. In this way, scheduled processes can continually receive the processor resource and advance through the system in predictably small steps rather than longer, often irregular ones.

For a time-sharing system to attempt to provide service to jobs that are highly interactive, it must not allow these jobs to remain in the system beyond times acceptable to the people initiating and accepting them; response time to process input must be short if the interaction is to proceed smoothly. The vast majority of the submitted jobs in a time-sharing system require short execution times and small amounts of memory. Hence, some memory must always remain free for the initiation of new tasks even at the expense of temporarily denying running processes further access to memory. The memory management algorithm in a time-sharing system should maintain an amount of available frames for running jobs and another amount uncommitted to these jobs, but available to newly arriving processes.

A. PROCESSOR SCHEDULING[4]

Processes active in the system exist on one of two queues used to control the scheduling of the CPU resource. A newly arrived task will be placed at the head of the single processor dispatching queue from which the next task to receive a CPU will be taken and will be given an initial time-slice value of approximately 27 milliseconds; such tasks are categorized as "neutral." All tasks begin executing their time-slices as neutral tasks regardless of the amount of virtual processing time that has elapsed for them. Once a task receives a processor, its time-slice running is stopped only upon the occurrence of a system-generated interrupt which is typically a page fault. Task-initiated interrupts (such as SVC's) are usually charged to the task's time-slice, but the system generated ones (page fault) are not. When a page fault occurs, the task is placed in the waiting queue and its time-slice is halted while the page is brought into memory by the PDP.

The page's arrival causes the task to be removed from the waiting queue and to be placed back at the head of the processor queue to make it immediately eligible for use of a processor. If the task exhibits a high paging rate during this initial time-slice (usually 25 page-read operations), the system recategorizes it as "privileged" if the number of privileged tasks is below the current system limit for such tasks; the task's time-slice is extended to 250 milliseconds when it is made privileged. However, should the number of privileged tasks existing in the system be at the limit, the task is categorized as "nonprivileged" and not allowed further access to the CPU's until a privileged task leaves that state.

Tasks leave the privileged state only through expiration of their time-slices or by termination. The expiration of the time-slice causes the task to be placed back in its original category as neutral with a new time slice of 27

milliseconds; it is then removed from its place on the processor queue and reinserted at the bottom.

B. MEMORY MANAGEMENT[4]

For the control of virtual memory pages belonging to tasks, UMMPS employs a page control block (PCB) for each page. With the occurrence of a page fault, the PCB for that page is placed at the end of the Page-in Queue (PIQ) and the PDP started if it is idle. The PDP schedules the page for reading either from the paging drum or the disk used for drum overflow conditions. When the PCB is removed from the PIQ, the PDP will request that the supervisor provide a memory frame for the incoming page. If the frame is received, a channel program is constructed by the PDP to read the page from its auxiliary storage address contained within the PCB. Should the supervisor refuse memory for the page, the PCB is placed on the Local Page-in Queue (LPIQ) for a later attempt to fill the memory request. The PDP searches the channel program of the drum for slots that have no outstanding page-read operations and requests pages to write onto the drum to fill these empty slots from the supervisor. The supervisor compares the number requested by the PDP with a computed need for writing.

In Version 2.2, the supervisor suspends writing of pages when the number of free memory frames and writing operations currently in progress exceeds fifteen. The supervisor selects the minimum of the PDP's requested amount and the number required to bring the available frame count up to fifteen and attempts to find this number of unreferenced pages from the Page-out Queue (POQ). The algorithm for selection is basically LRU with some minor modifications. Unreferenced pages are always selected, and referenced pages passed over in the search from the head to the end of the queue have their reference bits reset, thus making them likely to be taken during the next search.

[illegible]

The PDP receives the PCB's for writing and examines the change bits of each frame. Those frames that have not been changed will be returned to the supervisor for addition to the available memory supply, and those that have been modified will be scheduled for writing in one of the empty slot queues on the drum. At the conclusion of the writing operation, the reference bits are rechecked. If they have not been set in the frame, the frame is returned to the supervisor as available, otherwise, the page is put back at the end of the POQ.

Newly read pages of users' processes and system processes eligible for paging are added to the end of the POQ. Referencing of them by the PDP helps ensure that they are not immediately paged-out again pricr to their use by the requesting task. Since other tasks, possibly belonging to users, will receive the processor while a task is in a page-wait and will likely fault for pages as well, the PCB's on the POQ for a task are spread almost uniformly along its length. This lack of clustering of processes pages on the POQ diminishes the probability that a process will be paged-out completely by the present deallocation algorithm. Furthermore, the overhead incurred by examining the POQ for unreferenced pages does not appear to be substantial since the search lasts only long enough to find the required number, and resetting of the reference bits is limited to those passed over in the search.

The delay inherent in waiting for a page to be read or written by the drum system is minimally four milliseconds, and the least upper bound is 36 milliseconds which is associated with the page formatting used on the drum and the rotation speed of the mechanism. Further extensions to this minimum time for a page-read arise from three areas. The PDP is a process itself within the system and though it is not paged out, it does compete with other processes in the system for the CPU which is needed to build the channel programs that handle the page traffic. Secondly, the drum

must be "captured" or brought into synchronism with the building and execution of these programs; this delay can vary from zero to 17.5 milliseconds, or one physical rotation of the IBM drum. One rotation for all nine drum slots actually corresponds to two physical rotations of the mechanism. The last source of further delay is the length of the PIQ at the time the page fault occurs. Each entry ahead of a PCB in the PIQ decreases the probability that a memory frame will be available when the PDP requests it, and should it not be, an additional delay of 36 milliseconds plus some time in deallocation is incurred.

V. <u>ANALYSIS OF PFF IN THE UMMPS ENVIRONMENT</u>

As shown previously, the PFF decision algorithm is executed at the occurrence of a page fault. Since the supervisor runs with interrupts masked, time spent in the page fault handling routine delays the servicing of further interrupts. The page fault handler of the supervisor in Version 2.2 is already lengthy and the needed mechanism for PFF will tend to add an estimated time of five to ten percent that can only be recovered by a marked decrease in the overall frequency of page faulting. The two principal sources of this delay in execution for PFF lie in the computation and testing of the inter-page fault time or time-between-page-faults (TBPF) and the resetting of the task's reference bits which must occur at each fault.

A. THE TBPF

The evaluation of the TBPF is difficult. Few if any times will be optimal if the optimal value is established at only one point; hence, we shall establish an interval in which performance is considered optimal. Furthermore, the TBPF cannot be effectively "smoothed" without another increase in overhead for the system. The average times for the execution of IBM/360 instructions vary widely because as many as six different page references can be involved. Thus, scme means has to be incorporated to adjust statistically the TBPF in order that it be representative. Oscillaticn of the TBPF between being too low (gaining pages) and too high (losing them) has to be dampened either by adjusting the sample TBPF statistically or by increasing the sample interval. however, there may be an unacceptable increase in response time.

[illegible]

B. CONFLICTS FOR MEMORY

In a system such as UMMPS, user processes are permitted to expand their virtual memory beyond the physical capacity of the main memory. It is possible, though unlikely, that the working set of a process with poor locality, e.g. a list-processor, could consume all the allocatable memory of a PFF-controlled system without leaving its loading phase; alternatively, a number of smaller tasks could enter their loading phases in roughly the same period of time and exhaust the allocatable memory without lengthening their TBPF's to the point where they are required to reduce their working set sizes. In effect, there is some probability that the system could become deadlocked for memory and still remain within the bounds of the PFF algorithm. Therefore, a means must be incorporated within the system to detect this condition and force a resolution of the conflict for memory. Resolution can take one of two forms: either an adjustment to the processor scheduling algorithm is made or additions to the PFF algorithm are inserted.

The processor scheduling mechanism can suspend a number of the competing tasks at the point where deadlock occurs and allow the system to reclaim the memory allocated to their tasks for use by the nonsuspended ones. This reclamation activity will essentially halt user processing in the system until completion, and, unfortunately, it would occur just when the system load on memory and probably other resources was at its heaviest. The choice of which processes to halt would not be clear. Tasks that have small virtual memories (below the threshold for privileged status) usually represent interactive use; their delay would tend to reduce the interactive capability of the system.

Privileged tasks are processes which typically require much of the system's processor and virtual memory resources; the usual source of these is the batch input stream. Suspension of a batch-initiated task would probably yield a

greater amount of memory returned, but it would come at a significant increase in the cost of restarting the process later when the load was reduced. It seems more logical to detect the level at which the allocatable memory has reached a low point, e.g. at the present level of 15 frames, and suspend further starting of tasks until the level rises above this figure. This would reduce the probability that the system would reach a deadlocked state, but not totally eliminate it. Some empirically derived modifications to the suspension and restart levels could conceivably yield an extremely low probability of deadlock, but further experimentation is needed in this area.

Additions or modification to the PFF algorithm can be classified as either local to the process or global to all processes in the system. As shown previously, PFF is a local algorithm, and it is a view that holds that local process conditions can be so optimized as to eliminate global system problems. The deadlock over memory is a system-wide difficulty that can only be attacked indirectly by locally applied algorithms. A local modification to PFF would be to limit the maximum working set size of a process under all conditions of execution. If the present limit of 25 page-ins is set as a ceiling on the number of page frames allocatable to a process, then the processor and memory allocation schemes could effectively combine to partition these resources of the system between the shorter, small tasks usually associated with interaction and system overhead, and the longer, but less numerous, tasks coming through the batch input stream.

These latter tasks are identifiable by the supervisor, and the present system, a highly-modified version of HASP, is organized to limit initiation of them. The modification could take the form of applying the PFF algorithm as it presently stands until the task's memory reaches a certain level below the limit of 25. Once attaining this level, the task could request "privileged" assignment at which point it

is allowed to proceed to the limit of 25 pages; here it is forced from the loading state by compelling the deallocation of at least one frame for each page fault it has thereafter. Should all pages in the set be referenced, one might be selected purely at random. The system has no information other than that of the change bits; these indicate whether the selected page was unmodified. Since the overhead attendant to freeing unmodified pages is significantly less than for modified pages, "clean" pages would normally be freed. Reduction of the limit for privileged status to less than 25 page frames increases the likelihood that a task will request that status, and more tasks will be suspended pending privileged status. With the present structures, those suspended retain the scarce memory resource throughout their suspension; this memory could be forcibly reclaimed.

Leaving the neutral tasks to freely contend for memory can still allow deadlock to occur for no provision has yet been made to prevent a large number of small tasks from exhausting the available supply. Some help could come from the set of privileged tasks, though, since it is anticipated that PFF will eventually reduce their working sets to an amount much smaller than 25, and this reduction could be used to replenish the supply of available frames. This strategy amounts to requiring that privileged tasks fulfill their requirements for page frames from their own allocated areas as well as sustain any fluctuations in the working sets of neutral tasks.

Because the problem of memory management is global, the local solutions proposed above are not complete. Two possibilities exist here; augment the PFF algorithm by using the LRU technique to replenish the available frame supply when deadlock occurs, or allow the PFF algorithm to modify itself progressively as the supply diminishes. The first, using LRU, effectively suspends PFF since it has failed to allow the system supply to keep up with the process demand. As this demand increases, the likelihood that this LRU

algorithm will be called increases, and, therefore, the benefits of PFF will be offset. Systems with small memories such as that of NPS would be driven more easily and more often to this extreme, and the system overhead is increased by having to retain two algorithms for memory allocation. Comparable performance could probably be just as readily obtained by only one, namely LRU.

A more acceptable approach would be to allow the interval for optimal performance to be self-adjusting as conditions warrant by revising the location of the optimal TBPF interval according to system load. As the load on memory increases, the interval for optimal performance is moved progressively lower, thus causing more processes to appear to be operating optimally and to reduce the size of their working sets. When the load decreases, the interval is moved upward to allow them to accumulate larger working sets since the system than has the means to actively support their needs. This technique also serves to overccme a deficiency inherent to light system loading on memory.

In its present form, the PFF algorithm reduces a task's memory despite the fact that the system may not require the reduction in allocated frames. PFF attempts to maintain an established faulting rate regardless of system conditions, and allowing the interval for optimal performance to be moved according to the system load would make the algorithm adaptive and resfponsive to changing system conditions.

C. CHANGED PAGES

Another operating difficulty in the implementation of PFF is the problem of the changed page. The deallocation decision process of PFF calls for removing from memory those pages that have gone unreferenced since the last page fault by the process whenever the TBPF is high enough to warrant this action. Unchanged pages can be removed immediately and the frames added to the available supply, but changed or

"dirty" pages cannot be made available quickly, and their removal is a drain on the system's processing capability. They must be written onto the auxiliary storage medium so that the changed contents can be preserved for future use. This can be a source of considerable delay to the processing sequence in the system, since under heavy load conditions, processing must be effectively halted until the memory is made available.

The current UMMPS mechanism for writing could be extended to maintain a constant load that serves to "clean up" memory and raise the probability of having unchanged pages in the system. A user's task is heavily modified initially by the system in the loading and linking process of virtual memory construction, and all these modified pages must be written just as soon as the task proceeds into its execution phase and the TBPF drops sufficiently to allow the PFF allocation mechanism to take effect. Having been written onto the drum once, it is likely that the incidence of changed pages will drop as the user's process assumes control, but as yet no figures are available on program behavior to support this contention.

The present mechanism of the PDP for not freeing a page if it is referenced during the writing process prohibits a situation whereby a process could fault for a page that is actually available in memory, but it does so at some cost to the system. The current LRU algorithm writes pages only when the memory is needed, but PFF in its theoretical form would tend to give the PDP a heavy load for writing right after the loading process is complete. Some means could be fabricated to constantly change the fundamental composition of the POQ, but this would come at an increase in overhead to the system. The alternative, that of allowing the PDP to continuously write changed pages subject only to the slot capacity of the drum, would help keep main memory "clean", but the cost in system overhead for running the PDP process would be very high, and an increase such as this would have

the effect of significantly reducing the CPU cycles available for useful processing. In a memory limited system, the available excess processor capacity might well be applied to the task of making memory more readily available.

VI. A BASIC UMMPS IMPLEMENTATION OF PFF

A combination of the privileged task mechanism that limits task access to a processor and a revision to the PDP could produce a workable implementation of PFF within the basic UMMPS structures. Consider the present PIQ to be a single paging operation queue for the PDP. This queue, call it the Page Queue (PGQ), contains both pages to be read and pages to be written in an order noted below. The PDP executes the operation indicated by a new read/write bit within the PCB on this queue; all operations occur in the order of their appearance on the queue. The global POQ is not retained; however, each task maintains its own local equivalent called here, the LPOQ(i), for task(i). When a page is read for task(i), its PCB is put at the end of LPOQ(i).

The privileged task mechanism is modified as follows. If a task accumulates 25 page-ins while in a neutral state and its TBPF is still low, refusal of privileged status forces an end to its time-slice and resetting of all page reference bits. The usual time-slice ending mechanism applies here. Any other task which ends its time-slice before job termination does not reset its reference bits, but continues as presently implemented.

Any task faulting for a page will execute the PFF algorithm as previously shown in Section II.D. Any requirement for the task to release pages from its memory will result in the setting of the read/write bit in the affected PCB's. Pages to written will be removed from the LPOQ and placed ahead of the page to be read on the PGQ. The PDP's action is slightly modified at this point. PCB's are removed one at a time from the PGQ and scheduled for their appropriate operations. A page to be read must

contend for both memory from the supervisor and its slot address on the drum. Denial of either will result in the PCB's being placed on the Local Page Queue (LPGQ), which is an analogous version of the present LPIQ. The PDP will attempt to fill LPGQ requests prior to servicing requests from the PGQ just as it does now.

The supervisor, upon receiving a call for memory from the PDP, will answer first with free pages (it does sc now); should the supply be low (possibly zero), it searches the tasks on the processor queue from the bottom and examines the LPOQ's of the tasks. The algorithm for this search of each task's LPOQ will be roughly the same as for the present global POQ. Unchanged pages that are unreferenced will be freed immediately; changed, unreferenced pages will be placed on the PGQ for writing to the drum. Those pages passed over in the search will have their reference bits reset.

This organization is not without some difficulties. Forcing the writing of pages ahead of reading is sure to delay the restart of execution of the waiting tasks, but it could also result in an increased ability on the part of the system to handle new tasks. The overhead is higher than at present, but if the PFF algorithm is able to cope more effectively with a heavily loaded system than the LRU algorithm of the present version, it should be readily apparent upon testing.

VII. CONCLUSIONS

The PFF algorithm offers a useful technique for implementing the working set principle of Denning without the special hardware usually assumed to be needed for such a process. By using the TBPF as a working estimate cf the parameter tau, it overcomes some of the difficulties inherent in that principle, and it affords the system a means to estimate the size of a process's working set as it varies over time. The PFF algorithm does overcome the major difficulties of the LRU algoritm (thrashing) and appears to offer the same advantages that the working set principle has over LRU. PFF, like the working set principle, is a local algorithm; as such, it cannot overcome by itself the difficulties attached to a policy of unrestricted processor scheduling. Processor scheduling should be subordinate to memory allocation; once sufficient memory is available to contain the working set of a process, then a process should be allowed tc compete for the CPU resource. Unrestricted access to a processor compels acceptance of a global algorithm for memory allocation, and LRU appears to be the most viable of these.

UMMPS does not place any restrictions on the use of a processor by a task except in special cases; without a change to this policy, it will not be able to solve the operational difficulties inherent in the PFF algorithm nor fully attain the operational benefits. Any changes to the memory allocation scheme to achieve the goals of PFF can be expected to have a marginal effect; the LRU algorithm, or some other globally applicable deallocation policy, must still be retained to prevent deadlock over memory frames. Information is available within UMMPS to assign priorities to tasks, and this information could be used to assign and

[illegible]

control the allocation of memory as the task executes. A task that has received its allocation should be allowed to compete for a processor; those tasks that have not should be blocked until sufficient memory is available.

The present technique of processor scheduling is closely tied to the manner in which interrupts are handled in the system. Since interrupts and users' tasks are treated, in effect, in a like manner, it would cause an unacceptable delay in processing if initiation of a task were delayed due to the unavailability of memory; in the environment of UMMPS, memory is always made available to a requesting task by deallocation of other tasks' working sets if necessary. The aforementioned situations that limit PFF all arise from a shortage of memory frames to contain the working sets of the task's already in the system. The solution really appears to be either expansion of the system's physical main memory or a revision of the processor scheduling algorithm. In the words of Denning, "Paging is no substitute for real core"[6].

For a given system configuration, the best solution lies in a revision to the processor scheduling mechanism. This revision would require that a task not start until it has sufficient memory to contain its working set. The only tasks that do not have defined working sets are user originated, hence, the optimal solution would be to withold user tasks until the load on memory drops below an experimentally derived level. The PFF algorithm is an aid to estimating the working set, and a low TBPF is a warning to the sytem that the program's allocated memory is not sufficient to contain its working set just as a high TBPF is a signal that less memory can be used. The apparently optimal method is to withhold user tasks until the supply of available frames rises above a given level; in other words, subordinate processor scheduling to memory allocation as Kuehner and Randell[5] first recommended.

The method proposed in the previous section implements PFF within the existing environment of UMMPS, but with some modification to the queuing structures. It does retain an LRU algorithm to buttress PFF whenever the processor scheduling policy forces execution of tasks on a heavily loaded memory. The overhead in the method is substantially higher than the present method (modified LRU), but if the operational difficulties of PFF can be overcome, the benefits could more than compensate for this.

LIST OF REFERENCES

1 **Chu, W. Wesley, and Opderbeck, Holger,** "The Page Fault Frequency Replacement Algorithm," Volume 41, Part I, Pages 597-609, AFIPS Press, 1972.

2 **Denning, Peter J.,** "The Working Set Model for Program Behavior," Communications of the ACM, Volume 11, Number 5, Pages 323-333, May 1968.

3 **Mattson, R. L., and others,** "Evaluation Techniques for Storage Hierarchies," IBM Systems Journal, Volume 9, Number 2, 1970.

4 **Alexander, Michael T.,** "Time Sharing Supervisor Programs," Advanced Topics in Systems Programming, University of Michigan Engineering Summer Conferences, 7016, May 71.

5 **Kuehner, C.J., and Randell, B.,** "Demand Paging in Perspective," AFIPS Conference Proceedings, Volume 33, Part II, Pages 1011-1018, 1968.

6 **Denning, Peter J.,** "Virtual Memory," Computing Surveys, Volume 2, Number 3, Pages 153-190, September 1970.

7 **Watson, Richard W.,** Timesharing System Design Concepts, Pages 156-157, 167-174, McGraw-Hill, Inc., 1970.

8 **Morris, James B.,** "Demand Paging Through Utilization of Working Sets on the MANIAC II," Communications of the ACM, Volume 15, Number 10, Pages 867-872, October 72.

9 Users Manual, First Edition (with Change 20), William R. Church Computer Center, Naval Postgraduate School, Monterey, California, Page 1-7 (Figure 1.3), March 1970.

10 IBM System/360 Model 67 Functional Characteristics, Second Edition, International Business Machines Corporation, File No.S360-01, Form GA27-2719-1, January 1970.

11 IBM System/360 Principles of Operation, Eighth Edition, International Business Machines Corporation, File No. S360-01, Form A22-6821-7, September 1968.

12 **Gibson, Charles T.,** "Time-Sharing in the IBM System/360 Model 67," AFIPS Conference Proceedings, Volume 28, Pages 61-78, 1966.

13 **Alexander, Michael T.**, "Organization and Features of the Michigan Terminal System," AFIPS Conference Proceedings, Volume 40, Pages 585-591, 1972.

14 **Boettner, Donald W., and Alexander, Michael T.**, "MTS - Michigan Terminal System," SIGOPS Operating System Review, Volume 4, Number 4, Pages 6-19, December 1970.

15 University of Michigan Computing Center Memo #M200, KM & MTA/11-30-71, Page 3, December 1971.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Professor G. L. Barksdale, Code 72 Naval Postgraduate School Monterey, California 93940	1
4.	Capt Alexander Eugene Lancaster, Jr., USMC 129 Barberry Drive Berea, Ohio 44017	1
5.	Chairman, Computer Science Group, Code 72 Naval Postgraduate School Monterey, California 93940	1
6.	LTJG R. H. Bruebaker, Code 53Bh Naval Postgraduate School Monterey, California 93940	1

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)* Naval Postgraduate School Monterey, California 93940	2a. REPORT SECURITY CLASSIFICATION Unclassified
	2b. GROUP

3. REPORT TITLE

Implementation of the Page Fault Frequency Replacement Algorithm

4. DESCRIPTIVE NOTES *(Type of report and, inclusive dates)*

Master's Thesis; June 1973

5. AUTHOR(S) *(First name, middle initial, last name)*

Alexander E. Lancaster, Jr.

6. REPORT DATE June 1973	7a. TOTAL NO. OF PAGES 43	7b. NO. OF REFS 15

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO.	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Naval Postgraduate School Monterey, California 93940

13. ABSTRACT

This paper investigates the implementation of the page fault frequency (PFF) replacement algorithm as the mechanism for selecting and replacing pages of programs loaded into the main memory of a multiprocessing, multiprogrammed computer system. The frequency at which an executing program requires a page of virtual memory, the PFF, provides a basis for judging the real memory requirements of the program. Operating difficulties of PFF that reduce its usefulness in a time-shared computer system (Michigan Terminal System) are discussed, and a means of implementing the algorithm is proposed.

DD FORM 1 NOV 65 1473 (PAGE 1)

S/N 0101-807-6811

Security Classification

A-31408

Security Classification

KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Replacement Algorithms						
Virtual Memory						
Time-Sharing Systems						
Paged Memory						
Page Fault						
Michigan Terminal System						
MTS						

Thesis 145040
L2558 Lancaster
c.1 Implementation of the page fault frequency replacement algorithm.